



2.3 Built-in Exceptions

Exceptions can be class objects or string objects. Though most exceptions have been string objects in past versions of Python, in Python 1.5 and newer versions, all standard exceptions have been converted to class objects, and users are encouraged to do the same. The exceptions are defined in the module `exceptions`. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the `exceptions` module.

Two distinct string objects with the same value are considered different exceptions. This is done to force programmers to use exception names rather than their string value when specifying exception handlers. The string value of all built-in exceptions is their name, but this is not a requirement for user-defined exceptions or exceptions defined by library modules.

For class exceptions, in a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an ``associated value'' indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the `raise` statement. For string exceptions, the associated value itself will be stored in the variable named as the second argument of the `except` clause (if any). For class exceptions, that variable receives the exception instance. If the exception class is derived from the standard root class `Exception`, the associated value is present as the exception instance's `args` attribute, and possibly on other attributes as well.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition ``just like'' the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be sub-classed to define new exceptions; programmers are encouraged to at least derive new exceptions from the `Exception` base class. More information on defining exceptions is available in the *Python Tutorial* under the heading ``User-defined Exceptions.''

The following exceptions are only used as base classes for other exceptions.

exception `Exception`

The root class for exceptions. All built-in exceptions are derived from this class. All user-defined exceptions should also be derived from this class, but this is not (yet) enforced. The `str()` function, when applied to an instance of this class (or most derived classes) returns the string value of the argument or arguments, or an empty string if no arguments were given to the

constructor. When used as a sequence, this accesses the arguments given to the constructor (handy for backward compatibility with old code). The arguments are also available on the instance's `args` attribute, as a tuple.

exception `StandardError`

The base class for all built-in exceptions except `StopIteration` and `SystemExit`. `StandardError` itself is derived from the root class `Exception`.

exception `ArithmeticError`

The base class for those built-in exceptions that are raised for various arithmetic errors: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

exception `LookupError`

The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: `IndexError`, `KeyError`. This can be raised directly by `sys.setdefaultencoding()`.

exception `EnvironmentError`

The base class for exceptions that can occur outside the Python system: `IOError`, `OSError`. When exceptions of this type are created with a 2-tuple, the first item is available on the instance's `errno` attribute (it is assumed to be an error number), and the second item is available on the `strerror` attribute (it is usually the associated error message). The tuple itself is also available on the `args` attribute. New in version 1.5.2.

When an `EnvironmentError` exception is instantiated with a 3-tuple, the first two items are available as above, while the third item is available on the `filename` attribute. However, for backwards compatibility, the `args` attribute contains only a 2-tuple of the first two constructor arguments.

The `filename` attribute is `None` when this exception is created with other than 3 arguments. The `errno` and `strerror` attributes are also `None` when the instance was created with other than 2 or 3 arguments. In this last case, `args` contains the verbatim constructor arguments as a tuple.

The following exceptions are the exceptions that are actually raised.

exception `AssertionError`

Raised when an `assert` statement fails.

exception `AttributeError`

Raised when an attribute reference or assignment fails. (When an object does not support attribute references or attribute assignments at all, `TypeError` is raised.)

exception `EOFError`

Raised when one of the built-in functions (`input()` or `raw_input()`) hits an end-of-file condition (EOF) without reading any data. (N.B.: the `read()` and `readline()` methods of file objects return an empty string when they hit EOF.)

exception `FloatingPointError`

Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the **--with-fpectl** option, or the `WANT_SIGFPE_HANDLER` symbol is defined in the `pyconfig.h` file.

exception `IOError`

Raised when an I/O operation (such as a `print` statement, the built-in `open()` function or a method of a file object) fails for an I/O-related reason, e.g., ``file not found" or ``disk full".

This class is derived from `EnvironmentError`. See the discussion above for more information on exception instance attributes.

exception `ImportError`

Raised when an `import` statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.

exception `IndexError`

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not a plain integer, `TypeError` is raised.)

exception `KeyError`

Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception `KeyboardInterrupt`

Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function (`input()` or `raw_input()`) is waiting for input also raise this exception.

exception `MemoryError`

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C's `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception `NameError`

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

exception `NotImplementedError`

This exception is derived from `RuntimeError`. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method. New in version 1.5.2.

exception `OSError`

This class is derived from `EnvironmentError` and is used primarily as the `os` module's `os.error` exception. See `EnvironmentError` above for a description of the possible associated values. New in version 1.5.2.

exception `OverflowError`

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise `MemoryError` than give up). Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren't checked. For plain integers, all operations that can overflow are checked except left shift, where typical applications prefer to drop bits than raise an exception.

exception `ReferenceError`

This exception is raised when a weak reference proxy, created by the `weakref.proxy()` function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the `weakref` module. New in version 2.2: Previously known as the `weakref.ReferenceError` exception.

exception `RuntimeError`

Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong. (This exception is mostly a relic from a previous version of the interpreter; it is not used very much any more.)

exception `StopIteration`

Raised by an iterator's `next()` method to signal that there are no further values. This is derived from `Exception` rather than `StandardError`, since this is not considered an error in its normal application. New in version 2.2.

exception `SyntaxError`

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in an `exec` statement, in a call to the built-in function `eval()` or `input()`, or when reading the initial script or standard input (also interactively).

Instances of this class have atttributes `filename`, `lineno`, `offset` and `text` for easier access to the details. `str()` of the exception instance returns only the message.

exception `SystemError`

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

exception `SystemExit`

This exception is raised by the `sys.exit()` function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain integer, it specifies

the system exit status (passed to C's `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

Instances have an attribute `code` which is set to the proposed exit status or error message (defaulting to `None`). Also, this exception derives directly from `Exception` and not `StandardError`, since it is not technically an error.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `fork()`).

exception `TypeError`

Raised when a built-in operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

exception `UnboundLocalError`

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`. New in version 2.0.

exception `UnicodeError`

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of `ValueError`. New in version 2.0.

exception `ValueError`

Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

exception `WindowsError`

Raised when a Windows-specific error occurs or when the error number does not correspond to an `errno` value. The `errno` and `strerror` values are created from the return values of the `GetLastError()` and `FormatMessage()` functions from the Windows Platform API. This is a subclass of `OSError`. New in version 2.0.

exception `ZeroDivisionError`

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are used as warning categories; see the `warnings` module for more information.

exception `Warning`

Base class for warning categories.

exception `UserWarning`

Base class for warnings generated by user code.

exception `DeprecationWarning`

Base class for warnings about deprecated features.

exception `SyntaxWarning`

Base class for warnings about dubious syntax

exception `RuntimeWarning`

Base class for warnings about dubious runtime behavior.

Release 2.2.3, documentation updated on 30 May 2003.
See About this document... for information on suggesting changes.

PORTAL
US Patent & Trademark Office
Subscribe (Full Service) Register (Limited Service, Free) Login
Search: The ACM Digital Library The Guide
loop invariant range
THE ACM DIGITAL LIBRARY
Feedback Report a problem Satisfaction survey

Terms used **loop invariant range** Found **17,381** of **132,857**

Sort results by: relevance
Display results: expanded form
Save results to a Binder
Search Tips
Open results in a new window

Try an Advanced Search
Try this search in The ACM Guide

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next
Best 200 shown Relevance scale

1 Optimization of array subscript range checks
Jonathan M. Asuru
June 1992 **ACM Letters on Programming Languages and Systems (LOPLAS)**, Volume 1 Issue 2
Full text available: pdf(619.54 KB) Additional Information: full citation, abstract, references, citings, index terms, review

Compile-time elimination of subscript range checks is performed by some optimizing compilers to reduce the overhead associated with manipulating array data structures. Elimination and propagation, the two methods of subscript range check optimization, are less effective for eliminating global redundancies especially in while-loop structures with nonconstant loop guards. This paper describes a subscript range check optimization procedure that can eliminate more range checks than current meth ...

Keywords: conservative expression substitution, loop guard elimination, range check optimization

2 Session 18: dependence analysis/loop parallelization: The range test: a dependence test for symbolic, non-linear expressions
William Blume, Rudolf Eigenmann
November 1994 **Proceedings of the 1994 ACM/IEEE conference on Supercomputing**
Full text available: pdf(914.29 KB) Additional Information: full citation, abstract, references

Most current data dependence tests cannot handle loop bounds or array subscripts that are symbolic, nonlinear expressions (e.g. A (n*i+j), where $0 \leqslant j \leqslant n$). In this paper, we describe a dependence test, called the range test, that can handle such expressions. Briefly, the range test proves independence by determining whether certain symbolic inequalities hold for a permutation of the loop nest. Powerful symbolic analyses and constraint propagation techniques were developed to prove such in ...

3 Elimination of redundant array subscript range checks
Priyadarshan Kolte, Michael Wolfe
June 1995 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation**, Volume 30 Issue 6
Full text available: pdf(1.09 MB) Additional Information: full citation, abstract, references, citings, index terms

This paper presents a compiler optimization algorithm to reduce the run time overhead of

array subscript range checks in programs without compromising safety. The algorithm is based on partial redundancy elimination and it incorporates previously developed algorithms for range check optimization. We implemented the algorithm in our research compiler, Nascent, and conducted experiments on a suite of 10 benchmark programs to obtain four results: (1) the execution overhead of naive range check ...

4 Programming pearls: Writing correct programs



Jon Bentley
December 1983 **Communications of the ACM**, Volume 26 Issue 12

Full text available: pdf(648.55 KB) Additional Information: full citation, abstract

In the late 1960s people were talking about the promise of programs that verify the correctness of other programs. Unfortunately, it is now the middle of the 1980s, and, with precious few exceptions, there is still little more than talk about automated verification systems. Despite unrealized expectations, however, the research on program verification has given us something far more valuable than a black box that gobbles programs and flashes "good" or "bad"—we ...

5 A reexamination of "Optimization of array subscript range checks"



Wei-Ngan Chin, Eak-Khoon Goh
March 1995 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 17 Issue 2

Full text available: pdf(638.50 KB) Additional Information: full citation, abstract, references, citings, index terms, review

Jonathan Asuru proposed recently an enhanced method for optimizing array subscript range checks. The proposed method is however unsafe and may generate optimized programs whose behavior is different from the original program. Two main flaws in Asuru's method are described, together with suggested remedies and improvements.

Keywords: backward checks propagation, integer programming, loop guard elimination, safe bound checks optimization

6 Application of the goal invariant to the structuring of programs



Janusz W. Laski
January 1982 **ACM SIGPLAN Notices**, Volume 17 Issue 1

Full text available: pdf(774.02 KB) Additional Information: full citation, abstract, references

Functional rather than structural approach to conversion of a two-exit loop into a <u>while</u> loop is presented. The functions computed by the corresponding exits are formally specified and their union is analyzed to reveal the cause of its loop nonrealizability. The latter is removed by either redefining the domain/range of the constituent functions or by inclusion of the loop postludes into the loop specification. None of these methods changes the functionality of the original pr ...

Keywords: Structured programs, domain tuning, double exit loop, function, goal invariant, loop realizability, postludes, range tuning, termination assertion

7 Practical data breakpoints: design and implementation



Robert Wahbe, Steven Lucco, Susan L. Graham
June 1993 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation**, Volume 28 Issue 6

Full text available: pdf(1.37 MB) Additional Information: full citation, abstract, references, citings, index terms

A data breakpoint associates debugging actions with programmer-specified conditions on the memory state of an executing program. Data breakpoints provide a means for discovering program bugs that are tedious or impossible to isolate using control breakpoints alone. In practice, programmers rarely use data breakpoints, because they are either unimplemented or prohibitively slow in available debugging software. In this paper, we present the design and implementation of a practical data breakp ...

8 Heuristics for program synthesis using loop invariants
Joe W. Duran
January 1978 **Proceedings of the 1978 annual conference - Volume 2**

Full text available: pdf(535.95 KB) Additional Information: full citation, abstract, references, index terms

The problem of automatically synthesizing programs from formal specifications can be stated as, "constructively prove that there exists a program P which transforms valid input into output satisfying the required output assertion." If Prequires a loop, the proof will contain an induction. This is beyond the ability of current theorem provers and synthesis systems. In this paper, the loop invariant is used to reduce the problem of synthesizing loops to that of synthesizing three ...

Keywords: Automatic programming, Loop invariants, Program correctness, Program synthesis

9 Fusion-based register allocation
Guei-Yuan Lueh, Thomas Gross, Ali-Reza Adl-Tabatabai
May 2000 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 22 Issue 3

Full text available: pdf(475.45 KB) Additional Information: full citation, abstract, references, citings, index terms, review

The register allocation phase of a compiler maps live ranges of a program to registers. If there are more candidates than there are physical registers, the register allocator must spill a live range (the home location is in memory) or split a live range (the live range occupies multiple locations). One of the challenges for a register allocator is to deal with spilling and splitting together. Fusion-based register allocation uses the structure of the program to make splitting and spilling d ...

Keywords: performance evaluation, register allocation

10 Practical program verification: automatic program proving for real-time embedded software
John Nagle, Scott Johnson
January 1983 **Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages**

Full text available: pdf(739.44 KB) Additional Information: full citation, abstract, references

Despite the attractiveness of the concept, attempts to date to use proof of correctness techniques on production software have been generally unsuccessful. The obstacles encountered are not fundamental. We have implemented a proof of correctness system to be used for improving the realiability of certain small, real-time programs. It appears that many of the problems of past systems can be avoided.This work is supported by the Long Range Research Program of the Ford Motor Company, Dearborn, Mich ...

11 A fresh look at optimizing array bound checking
Rajiv Gupta
June 1990 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1990 conference**

on Programming language design and implementation, Volume 25 Issue 6

Full text available: pdf(998.53 KB) Additional Information: full citation, abstract, references, citings, index terms

This paper describes techniques for optimizing range checks performed to detect array bound violations. In addition to the elimination of range checks, the optimizations discussed in this paper also reduce the overhead due to range checks that cannot be eliminated by compile-time analysis. The optimizations reduce the program execution time and the object code size through elimination of redundant checks, propagation of checks out of loops, and combination of multiple checks into a single c ...

12 Subsequence references: first-class values for substrings

Wilfred J. Hansen

October 1992 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 14 Issue 4

Full text available: pdf(1.31 MB) Additional Information: full citation, abstract, references, citings, index terms, review

Arrays of characters are a basic data type in many programming languages, but strings and substrings are seldom accorded first-class status as parameters and return values. Such status would enable a routine that calls a search function to readily access context on both sides of a return value. To enfranchise substrings, this paper describes a new data type for substrings as a special case of one for general subsequences. The key idea is that values are not sequences or references to positi ...

Keywords: ATK, AUIS, Andrew Toolkit, Ness, document processing, programming language design, sequences, string searching, strings, subsequences, substrings

13 Dynamically discovering likely program invariants to support program evolution

Michael D. Ernst, Jake Cockrell, William G. Griswold, David Notkin

May 1999 **Proceedings of the 21st international conference on Software engineering**

Full text available: pdf(1.59 MB) Additional Information: full citation, references, citings, index terms

Keywords: dynamic analysis, execution traces, formal specification, logical inference, pattern recognition, program invariants, software evolution

14 On defining application-specific high-level array operations by means of shape-invariant programming facilities

Sven-Bodo Scholz

July 1998 **ACM SIGAPL APL Quote Quad , Proceedings of the APL98 conference on Array processing language**, Volume 29 Issue 3

Full text available: pdf(583.03 KB) Additional Information: full citation, abstract, references, citings, index terms

Most of the existing high-level array-processing languages support a fixed set of pre-defined array *operations* and a few higher-order functions for constructing new array operations from existing ones. In this paper, we discuss a more general approach made feasible by SAC (for Single Assignement C), a functional variant of C.SAC provides a meta-level language construct called WITH-loop which may be considered a sophisticated variant of the FORALL-loops ...

Keywords: compilation, high-level array operations, meta-level programming, performance comparison, shape-invariant programming

15 Structured Programming with go to Statements
Donald E. Knuth
December 1974 **ACM Computing Surveys (CSUR)**, Volume 6 Issue 4
Full text available: pdf(3.02 MB) Additional Information: full citation, references, citings, index terms

16 Debugging highly-optimized Ada with code motion (DHACM)
Kevin Tucker
November 1997 **Proceedings of the conference on TRI-Ada '97**
Full text available: pdf(721.40 KB) Additional Information: full citation, references, index terms

17 Optimization of expressions in Fortran
Vincent A. Busam, Donald E. Englund
December 1969 **Communications of the ACM**, Volume 12 Issue 12
Full text available: pdf(1.19 MB) Additional Information: full citation, abstract, references, citings, index terms

A method of optimizing the computation of arithmetic and indexing expressions of a Fortran program is presented. The method is based on a linear analysis of the definition points of the variables and the branching and DO loop structure of the program. The objectives of the processing are (1) to eliminate redundant calculations when references are made to common sub-expression values, (2) to remove invariant calculations from DO loops, (3) to efficiently compute subscripts contain ...

Keywords: DO loops, FORTRAN, common subexpressions, compilation, compilers, expressions, invariant calculations, optimization, register allocation, subscripts

18 Effective sign extension elimination
Motohiro Kawahito, Hideaki Komatsu, Toshio Nakatani
May 2002 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation**, Volume 37 Issue 5
Full text available: pdf(389.09 KB) Additional Information: full citation, abstract, references, index terms

Computer designs are shifting from 32-bit architectures to 64-bit architectures, while most of the programs available today are still designed for 32-bit architectures. Java™, for example, specifies the frequently used int" as a 32-bit data type. If such Java programs are executed on a 64-bit architecture, many 32-bit values must be sign-extended to 64-bit values for integer operations. This causes serious performance overhead. In this paper, we present a fast and effective algorithm for e ...

Keywords: 64-bit architectures, IA-64, JIT compilers, Java, sign extension

19 Automatic program specialization for Java
Ulrik P. Schultz, Julia L. Lawall, Charles Consel
July 2003 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 25 Issue 4
Full text available: pdf(1.18 MB) Additional Information: full citation, abstract, references, citings, index terms

The object-oriented style of programming facilitates program adaptation and enhances

program genericness, but at the expense of efficiency. We demonstrate experimentally that state-of-the-art Java compilers fail to compensate for the use of object-oriented abstractions in the implementation of generic programs, and that program specialization can eliminate a significant portion of these overheads. We present an automatic program specializer for Java, illustrate its use through detailed case stud ...

Keywords: Automatic program specialization, Java, object-oriented languages, optimization, partial evaluation

20 Program optimization and parallelization using idioms



Shlomit S. Pinter, Ron Y. Pinter

January 1991 **Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**

Full text available: pdf(1.23 MB) Additional Information: full citation, references, citings, index terms

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next

The ACM Portal is published by the Association for Computing Machinery. Copyright © 2004 ACM, Inc.
Terms of Usage Privacy Policy Code of Ethics Contact Us

Useful downloads: Adobe Acrobat QuickTime Windows Media Player Real Player